OpenKnowledge

FP6-027253

OpenKnowledge Visualiser Sub-Components

David Dupplaw[1], Sindhu Joseph[2], Paolo Besana[3], Madalina Croitoru[1], Srinandan Dasmahapatra[1], Bo Hu[1], Paul Lewis[1], Antonis Loizou[1], Liang Xiao[1]

[1] IAM Group, School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK.
[2] Artificial Intelligence Research Institute, CSIC (Spanish Scientific Research Council), Campus Universitat Autonoma de Barcelona, 08193 Bellaterra, Catalonia, Spain.
[3] Centre for Intelligent Systems and their Applications, University of Edinburgh, Edinburgh, EH8 9LE, UK.

Report Version: final
Report Preparation Date: December 2007
Classification: deliverable 5.4
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

OpenKnowledge Visualiser Components: Visual Diagnostic and Visual Authoring Tools

David Dupplaw Sindhu Joseph Paolo Besana
Madalina Croitoru Srinandan Dasmahapatra Bo Hu
Paul Lewis Antonis Loizou Liang Xiao

January 2, 2008

Abstract

OpenKnowledge is based on the creation of interaction models that run across a distributed peer to peer network. Although the learning curve to creating interaction models has been minimised through the use of a simple process language, it is clear that a graphical representation of an instance of an execution of an interaction model would be very useful in diagnosing logical errors in the interaction model. In this deliverable we explain the diagnostic visualiser tool that provides a means for receiving information about the execution of an interaction model. We also describe our work on the development of visual composition tools for the creation of interaction models in LCC. In particular we describe an algorithm that provides a conversion from Electronic Institutions (EIs) to Lightweight Coordination Calculus (LCC) allowing the utilisation of existing graphical composition tools for EIs.

1 Introduction and Motivation

In computer programming, there are two types of programming error. The first is syntactic errors that usually cause the language compiler to throw an error when compilation is attempted. These can be caught early when the cost of correction is minimal. In OpenKnowledge, the language used to develop interaction models is LCC [2]. The parser we have developed for LCC is able to detect syntactic errors and provide some assistance in identifying the errors. This tool is included as part of the standard OpenKnowledge user interface.

The second type of programming error, the logical error, is often known coloquially as a bug. Computer programmers have debuggers that help with the correction of these errors. Debuggers are able to hook into the program execution and catch points at which the execution fails. This process is made easier because the execution is contained within one machine. Logical errors in interaction models can be costly because they change the state of the network and can be more difficult to track down. Executions of interaction models may

fail due to problems or restrictions on other peers of which the author of the interaction model has no control. This can make it difficult to detect whether failure was due to logical errors or some other factors.

In the following sections we will introduce the tools that are in development to aid the diagnostics of interaction models. We will also introduce our work in creating visual composers that allow interaction models to be programmed graphically.

2 Local Execution Framework

During the creation of the OpenKnowledge kernel, the developers of the kernel required a means for testing and executing the correctness of the code which executes interaction models. Having to create networks when the code-base was so immature meant that debugging the kernel code was difficult.

To deal with these problems, a framework was created that allows the execution of interaction models on a single computer, without the need to access the OpenKnowledge network. It simulates the interaction by creating many peers within the one single machine - enough to fulfil all the roles within the interaction model. It then invokes the execution of the interaction model using a coordinator of which the local framework has control. The main advantage of the framework is that it still utilises the kernel's adapter, interpreter and coordinator modules, thereby providing a realistic simulation of the actual execution of the model.

This framework is a very useful tool for the author of interaction models. Placing an interaction model onto the network which has never been run can be a worrying experience for an author - how do they know if they got it right? By executing the ineteraction model locally, the author can ensure that the logic is correct for the various possibilities of the interaction model. With some development, the framework could provide automatic hooks into constraints in the model allowing the author to test different paths in the model by forcing constraints to succeed or fail.

3 Execution Visualisation Framework

Visualisation of the execution of an interaction model can be a useful tool for the identification of failure points in the interaction. Peers which have not responded, or have responded in an unexpected way, can be identified much more easily.

As the coordinator peer has control over the execution of the interaction model, it is able to know the state of the interaction at every point. It knows which peers have failed to satisfy constraints, which messages have been sent and which branches in the interaction model have been taken. This data is very useful for interaction model authors who are testing their models.

To retrieve this information and present it to a user, the kernel has a diagnostic module that connects a peer with a coordinator peer. That coordinator peer will execute interaction models and will inform connected peers, that wish to be informed of the diagnostic information, when messages are sent and received and when constraints are solved. This follows the standard event-listener design pattern but in a distributed manner.

The diagnostic module comprises of a set of messages and a component that is able to handle those messages. The state of the diagnostics for many simultaneous interactions can be handled such that a visualisation can access the information for a specific interaction run and create a display. This fits the standard model-view-controller (MVC) design pattern, where the coordinator is the controller, the model is stored in the diagnostic module and the visualisation of the diagnostics is the view.

There are two means by which a peer can register itself with the coordinator to receive the diagnostic information. The first is where a peer registers itself with a coordinator to receive diagnostic information for an interaction that is already running. The second is to register itself during the bootstrap process. The latter registration method is necessary for peers that wish to examine the real-time execution of the interaction model, whereas the former is useful for diagnosing interactions that have, for example, been running longer than expected. Note that to receive information about a running interaction, the peer must be aware of the peer identifier of the coordinator peer and must also be aware of the identifier of the interaction run on that coordinator.

The coordinator may have policies that might disallow a peer from being sent diagnostic information. These policies may be based on peer configuration, physical limitations of the peer or from values retrieved through the trust and reputation service. An example restriction might be that only peers that are playing within an interaction may receive the diagnostics.

The registration with the coordinator may include information that informs the coordinator that the peer only wishes to receive certain types of events that occur within the interaction. For example, a peer may only be interested in events where the satisfaction of constraints time-out. The types of events that have been identified so far are as follows:

- **Message Sending:** Fired when an agent in the interaction sends a message.
- **Message Reception:** Fired when an agent in the interaction receives a message. Note that the agent may not handle this message or be exepcting the message.
- **Message Consumption:** Fired when an agent in the interaction consumes a message.
- **Constraint Satisfaction Request:** Fired when an agent requires a constraint to be solved.
- **Constraint Satisfied:** Fired when a constraint is satisfied.

- **Constraint Failure:** Fired when a constraint fails to be satisfied.
- **Constraint Message Sent:** Fired when the coordinator sends a message to a peer to satisfy a constraint.
- **Constraint Message Received:** Fired when the coordinator receives a message from a peer that has been satsifying a constraint.
- **Coordinator Select-Peers Request:** Fired when the coordinator asks a peer to select a compatible list of peers to play with.
- **Coordinator Select-Peers Response:** Fired when the coordaintor receives a select-peers response from a potential peer in the interaction.
- **Committment Request Sent:** Fired when the coordinator asks a peer to commit to the interaction
- **Committment Request Received:** Fired when the coordinator received a committment request from a peer.

The diagnostic module is not limited to those interaction events above, so a peer must also perform filtering of incoming events and this can be achieved by registering the interaction events with event handlers. The communication layer handles message events by invoking specific handlers for those events. The handler for the diagnostic messages (the diagnostic module) will be responsible for redirecting the specific interaction event to an interaction event handler. Again, this can be provided by the standard event listener design pattern.

Note that the above list of interaction events are fired within the coordinator, such that the coordinator can inform listening peers. However, the firing of such an event within the coordaintor does not necessarily result in a diagnostic message being sent. The reason is that some events occur before peers may have joined the interaction. In cases like this the event will be logged in the coordinator. This diagnostic log may be delivered to a peer when it is able to receive diagnostic information. For example, the select-peers request/response events will not be sent to a peer because peers cannot register for diagnostic information before the committment to play the interaction is made.

3.1 Getting Diagnostics for a Running Interaction

A peer that wishes to be informed of the diagnostic information for an interaction that is already running, must register itself as a listener with the coordinator peer. This comprises a request and response message to and from the coordinator.

The coordinator will receive requests for listener registrations and will accept those that fit within its policy rules. Those peers will receive successful registration responses, while others will receive failure responses such that they understand that they will not be receiving diagnostic information.

Once the diagnostic link has been made, the coordinator will send messages to the listener peer when events, for which the peer has registered, are fired.

The coordinator will not expect response messages - the messages are delivered with no guarantee of receipt. This minimises network overhead.

The coordinator may be configured such that it will log all events for the duration of an interaction. This will allow joining peers to receive a complete history of the interaction, up to its current state, without having to listen from the start. However, it is important that logging is user configurable, as this will incur a processing overhead that may be unacceptable.

3.2 Getting Diagnostics From Interaction Initialisation

During bootstrapping of an interaction, after a mutually compatible group of peers has been chosen, all peers that will play in the interaction are contacted to commit to play the interaction. This commitment can provide a means for a peer to flag that it also wants to receive diagnostic information. Clearly, this disallows peers outside of the interaction from receiving the information from the start, however, it is expected that the peers that will want to receive the "live" diagnostic information will be a members of the interaction.

When the coordinator receives a commitment request with the additional request for diagnostics, it will register the peer in its diagnostic listeners list for the interaction run that is being committed to. Thereafter, the coordinator will send messages to the listener peer when events, for which the peer has registered, occur.

3.3 Coordinator Support

For the diagnostic module to be informed of the necessary events that occur in the interaction model, the coordinator and its interpreter need to be augmented with some extra features.

These features are:

- **Interaction Run Identifiers:** The coordinator must be able to uniquely identify each interaction model run that it is handling, such that incoming registration requests can be directed to the correct interpreter. It also allows the peer receiving the diagnostics to be able to correctly handle the incoming messages if multiple interaction instances are sending diagnostic data to it. The unique identifier can be created based on the coordaintor's peer ID and a unique number which the coordaintor generates for a run. This will ensure the global uniqueness of the interaction run identifier.

- **Interpreter Events:** So that the coordinator is able to inform registered peers of events that occur in the interaction model, the interpreter must be abnle to produce interaction events and inform the coordinator. The best method of doing this is to apply the event listener design pattern to the communication.

- **Coordinator Events:** Some diagnostic events are generated by the coordatinor meaning the coordinator code needs to be extended to fire the

events. Such events include the reception of a constraint message.

- **Further Message Types:** On receiving an event from the interpreter, the coordinator needs to determine the appropriate peers that need to be informed (based on their event filters) and send a message to them informing them of the event. The message will be a single message, but must be capable of delivering different event types. The type of event that occured in interpretation is encoded in the message. The possible event types were described in section 3.

4 Diagnostic Visualisation

The use of the event listener pattern for diagnostic information means that the visualisation of the diagnostic information can be provided by any means. The API provided by the listener means that any object that implements the API can provide a visualisation.

To demonstrate the use of the diagnostic visualiser, a sequence diagram component has been provided for the user interface. This is a listener for some diagnostic events - namely the message sending and receiving between agents in the interaction model.

The visualisation is based on a generic graphical class that we have developed that is able to draw standard sequence diagrams.

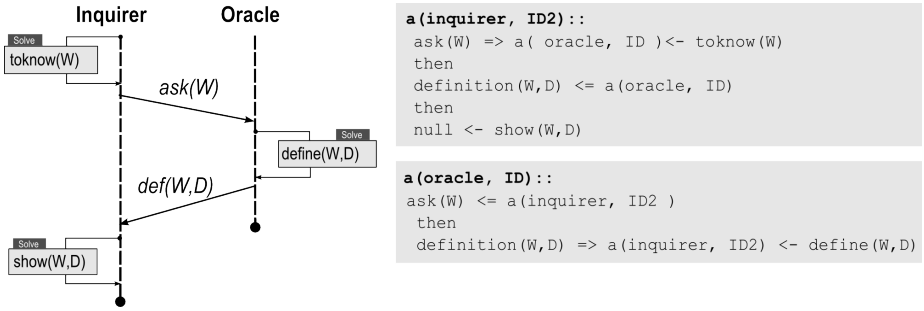


Figure 1: An interaction model and its diagrammatic representation

At the time of writing the visualisation is still under development, so Figure 1 shows an example of the expected interaction visualisation for an interaction model. Along the top of the diagram the participants that are interacting in the interaction model are shown. The vertical line from them represents their timeline. Message operations are drawn between the participants in solid lines when they are consumed at the receiver. The type of the message is shown along the message line.

It is possible to toggle the display of the the coordinator as an agent in the interaction (in a different colour to differentiate it from the agents defined in the

interaction model). The messages to and from the coordinator (for constraint satisfaction requests and responses) are drawn in red.

5 Interaction Model Visual Composition from Electronic Institutions

An aspiration of OpenKnowledge is to allow users who have very little knowledge of the underlying technology to use the system effectively. To that end, it is important that some tools are developed that allow users to create interaction models without the need to learn and program LCC and Java.

As part of their work on Electronic Institutions (EIs), IIIA-CSIC at the Universitat Autonma de Barcelona have created graphical interface tools for their creation. The Electronic Institutions Development Environment (EIDE) is a set of tools aimed at supporting the engineering of intelligent distributed applications as electronic institutions. The EIDE is composed of:

- *Islander* A graphical tool that supports the specification of the rules and protocols in an electronic institution.
- *SIMDEI/OMS* Simulation tool to animate and analyse specifications created with Islander prior to the deployment stage.
- *Agent builder* Agent development tool.
- *AMELI* Software platform to run electronic institutions. Once an electronic institution is specified with Islander is ready to be run by AMELI without any programming.

As EIs constitute a super-set of the basic LCC functionality, non-complex EIs may be converted into LCC. The use of the existing user interface for EIs provides a lead-in for current users of Islander to use the OpenKnowledge platform, as well as leveraging the work put into creating the Islander graphical interface. In the following sections we describe the functionality of EIs before describing how these may be converted into LCC.

5.1 Electronic Institutions

Loosely speaking, Electronic Institutions (EIs) are computational realizations of traditional institutions; that is, coordination artifacts that establish an environment where agents interact according to stated conventions, and in such a way that interactions within the (electronic) institution would count as interactions in the actual world.

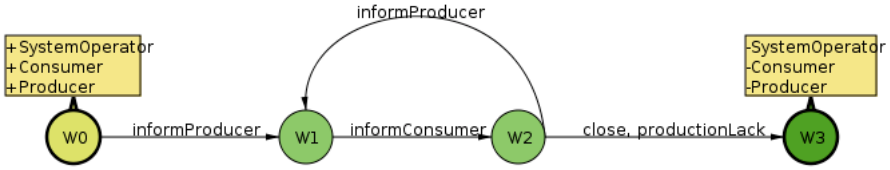
According to the basic definition of an electronic institution [1], an EI is composed of three components: a *dialogical framework* that establishes the social structure, the ontology, and a communication language to be used by participating agents (playing either institutional staff or non-institutional external

roles); a *performative structure* defining the activities (also named scenes) along with their relationships; and a set of *norms* defining the consequences of agents' actions.

5.1.1 Scene protocols

A scene is, in broad terms, a conversation protocol played by a group of agents. More precisely, a scene defines a generic pattern of conversation protocol between roles. Any agent participating in a scene has to play one of its roles. It is generic in the sense that it can be repeatedly played by different groups of agents. In the same sense that the same theater scene can be performed by different actors incarnating the same scene characters.

A scene protocol is specified by a finite state directed graph where the nodes represent the different states of the conversation and the directed arcs connecting the nodes are labeled with the actions that make the scene state evolve. These are: illocution schemes and timeouts. The graph has a single initial state (non-reachable once left) and a set of final states representing the different endings of the conversation. There is no arc connecting a final state to some other state. The next image is an example of a scene protocol.




```
W0 -> W1 [ informProducer ]: (inform (?so SystemOperator) (?p Producer) ?contract:Contract)
   <=> ( ?p == ?contract.producer ) && ( ?contract.price > 0 ) && ( ?contract.quantity > 0 )
W1  -> W2 [ informConsumer ]: (inform (!so SystemOperator) (?c Consumer) !contract)
   <=> ?c == !contract.consumer
   => marketDemand = marketDemand - !contract.quantity
      marketOffer = marketOffer - !contract.quantity
      marketContracts = marketContracts + [!contract]
W2  -> W1 [ informProducer ]: (inform (!so SystemOperator)(?p Producer) ?contract)
    <=> ( ?p == =contract.producer) && ( ?contract.price > 0 ) && ( ?contract.quantity > 0 )
W2 -> W3 [ close ]: (inform (!so SystemOperator) (all all) close() )
    <=> marketDemand <= 0
W2 -> W3 [ productionLack ]: (inform (!so SystemOperator) (all all) productionLack() )
    <=> ( marketDemand > 0 ) && ( mrketOffer < 0 )
```

5.1.2 Illocutions

An *illocution schema* is part of the transition definition. It defines the conditions necessary for an illocution to produce a state change, and the transition's resulting side actions.

Illocutions schemas are defined as $IS = \iota((ag_s, R_s), (ag_r, R_r), \varphi)$. Where ag_s and R_s are the sending agent identifier and role, ag_r, R_r are the receiving agent's identifier and role, ι is the illocutionary particle and φ is the propositional

content being sent. Illocutions may contain variables which give the engineer greater specification power.

By *illocution* we refer to those illocution schemas whose variables have been grounded. To say it in another way, an illocution is an illocution schema whose definition is composed exclusively by constants. Grounding is the process through which an illocution schema becomes an illocution by substituting the variables by their actual values.

Variables can be used for illocution schema specification. They can appear as bound or free. A free variable v_f is grounded by matching it with any value of its type. Then, the value 'c_{n+1}' used to match v_f is added as a new binding $b_i = (v_f, c_{n+1})$ to σ in the execution environment. A variable is specified as free when prefixed with '?'. On the other hand, when a schema with a variable v_b, specified as bound, is matched against an actual illocution, it can only be substituted by the value 'c_n'. Where 'c_n' is v_b's last binding $b_l = (v_b, c_n) \in \sigma$. A variable is specified as bound when prefixed with '!'. In other words, if the value being matched against '!v_b' is different from 'c_n', the illocution uttered is unacceptable.

Most parts of an illocution schema can be specified with variables: agent ids, agent roles, and propositional content. Constraints, and actions labeling arcs can also use variables although only as bound.

5.2 Converting EIs to LCC

The Electronic institution is a state based model as it keeps track of the state of the conversation at any instant of time and defines a conversation as a movement between states. LCC on the other hand views a conversation from each of the agent roles point of view. The state of the conversation is implicit in LCC and is represented by the sequence and parallel operators. Thus a translation from EI to LCC is essentially a translation from the state based view to a role based view. That is equivalent to translating an EI scene to a collection of scene projections corresponding to each of the roles in the scene[1].

The translation generates one LCC clause for each role state combination. Inside the clause the LCC code refers to what an agent of the given role can do in a given state. The conversation thread can be recovered by the *"then"* operator which works as a transition from one state to the next for the given role. Inside each clause the algorithm checks for the possible transitions out of that state and each of such transitions is translated connecting them with an *"or"* clause. If the transition's end state is a final state then the clause ends otherwise the translation is called again.

[1] For each agent variable and role in a scene specification, we can obtain its scene projection. $S_{r_1} = (R, DF, w_0, W_f, (WA_r)_{r\in R}, (WE_r)_{r\in R}, \theta_{r_1}, \lambda, min, Max)$ is the projection of scene $S = (R, DF, w_0, W_f, (WA_r)_{r\in R}, (WE_r)_{r\in R}, \theta, \lambda, min, Max)$ for agent role r_1. Where $\theta_{r_1} = \{(w_i, w_j) \in W \times W | \lambda(w_i, w_j) = \iota((\_, R_s), (\_, R_r), \_)\ and\ ((R_s = r_1)\ or\ (R_r = r_1))\}$. The intuition behind a scene projection is that when you project a scene through a role, you end up with a 'sub-scene' where the agent role is present in all the illocution schemes. The projected scene may be an unconnected graph.

The main algorithm extracts the set of transitions of an EI scene. Then for each of those transitions it invokes an *extractLCC* method once for the sender and then for the receiver agent roles of the transition, the results of which are appended to the LCC clause for the sender role and the receiver role respectively. If there is not yet a role, it first invokes a *createRole* method and then appends the clause as above.

ExtractLCC translates a given EI transition to an LCC Clause. The output contains an LCC clause for the illocution defined by the transition. One of the parameter to this method is the role information. The generated LCC clause varies depending on whether it is invoked with a sender or a receiver role.

The transformation algorithm works on the following assumptions:

- Each transition must be labeled with only one illocution.
- Each agent variable must only be used for agents of one and only one role.
- There is only one entry and exit state for all agent roles.

Previous to executing the transformation algorithm on a scene, its specification needs to be transformed to a scene that has the said properties. The first property is attained by getting the transitions with more than one illocution schema and generate one transition for each illocution schema. All of the generated transitions will have the same start and end state, and each will have one of the illocution schemas of the original transition. The second property is attained by renaming those agent variables that are used for agents of more than one role. The renaming is done by appending the role ID to the variable name, generating as many different agent variables as agent roles it represented. The third property is introduced by permitting only such scenes at the input level.

5.2.1 Use of LCC operators

In order to define a message passing protocol, LCC has three operators; *"then"* , *"par"*, and *"or"*. The operators define sequence, parallelism, and choice respectively. Since EI scenes are played sequentially (only one agent speaks at a time), the transformation will not generate the "par" operator. The use of the other two operators is explained below.

An EI scene is sequential, every agent can only be in one state at a time. This means that the transitions from a state should be represented as possibilities with only one of them realizable at a time. If there are more than one possible transitions from an EI state, then the corresponding LCC code uses *"or"* operator to connect the clauses generated out of those transitions. Note that the *"or"* operator is used only if at least one role appears as *sender* or *receiver* in more than one transition. Also it should be noted that the LCC code parts connected with an *"or"* operator are mutually exclusive, if one happens the others cannot.

The *"then"* operator is used to indicate sequence in the protocol. As EI scenes are sequential, moving from one state to the next is realized through the

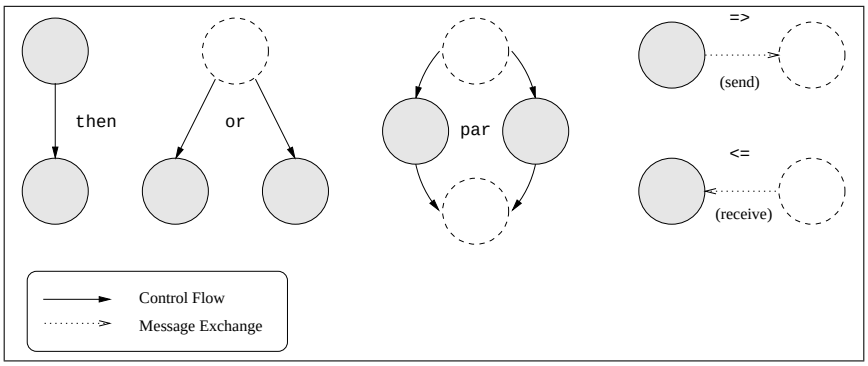


Figure 2: LCC operators

"then" operator in the translation. It acts as a connective to link a transition to its target state. The LCC operators are shown in figure 2.

5.2.2 Illocution schema translation

The method ExtractLCC translates the illocution schema to the appropriate LCC clause. ExtractLCC first checks if the caller is a receiver role. If yes, it simply generates the schema "$msg \Leftarrow sender$" where sender is elaborated with sender role, sender variable and the variables available in the illocution schema.

ExtractLCC if called with the sender role, checks for the constraints in executing the illocution prior to generating the schema "$msg \Rightarrow receiver$". Constraint checking is done by extracting the constraints and evaluating them. Some of the values that will always need to be checked are the bound variables (see 5.2.3).

5.2.3 Variables

A major difference between EI scene and the corresponding LCC protocol is that an EI scene has a global state where it can keep track of the state of the scene variables as the scene execution progresses. Where as the corresponding LCC variables have local scope restricted to a clause. Thus it becomes necessary to ensure references to the same variables and values across LCC clauses by other mechanisms. Here it is realized through variable passing. The variables are picked up as and when encountered in an illocution and passed as parameters of agent roles to the next clause. This way the scope of the variables is extended to the boundaries of a scene.

5.2.4 Constraints $actions \leftarrow message \leftarrow constraints$

The constraints contain the operations to extract data from the scene and other arithmetic and logic operations. The actions contain operations to update vari-

ables of the scene, in addition to those operations that can be used in the constraints.

Below we specify the algorithm details. We have used certain procedure names which are self explanatory but are not defined. It is straightforward to define those small functions and so we have not included them in the document.

Algorithm 1 EI2LCC(*scene*)

```
 1: lcc ← NIL
 2: for transition ∈ getTransition(scene) do
 3:     for roles ∈ getRoles(lcc) & states ∈ getStates(lcc) do
 4:         if equals(roles, senderRole) & equals(states, source) then
 5:             lcc⟨senderRole,source⟩ ← or + extractLCC(transition, "sender") + then +
                addTargetClause(target.variables)
 6:             senderFound ← 1
 7:         end if
 8:         if equals(roles, receiverRole) & equals(states, source) then
 9:             lcc⟨receiverRole,source⟩ ← or + extractLCC(transition, "receiver") + then +
                addTargetClause(target.variables)
10:             receiverFound ← 1
11:         end if
12:     end for
13:     if !senderFound then
14:         lcc⟨senderRole,source⟩ ← newRoleClause(clause, lcc_role, source)
15:         lcc⟨senderRole,source⟩ ← extractLCC(transition, "sender") + then +
            addTargetClause(target.variables)
16:     end if
17:     if !receiverFound then
18:         lcc⟨receiverRole,source⟩ ← newRoleClause(clause, lcc_role, source)
19:         lcc⟨receiverRole,source⟩ ← extractLCC(transition, "receiver") + then +
            addTargetClause(target.variables)
20:     end if
21:     lcc⟨senderRole,target⟩ ← newRoleClause(clause, lcc_role, target)
22:     lcc⟨receiverRole,target⟩ ← newRoleClause(clause, lcc_role, target)
23: end for
24: return lcc
```

Algorithm 2 extractLcc(*trans*, *role*)

```
    lcc_send ← NIL
 2: if role = "sender" then
        lcc_send ← lcc_send + actionsToLcc(trans.actions)
 4:     lcc_send ← lcc_send + messageToLcc(trans.message, trans.source.variables)
        lcc_send ← lcc_send + constraintsToLcc(trans.constraints)
 6: end if
    lcc_rec ← NIL
 8: lcc ← lcc_send
    if role = "receiver" then
10:     lcc_rec ← messageToLcc(trans.message)
        lcc ← lcc_rec
12: end if
    return lcc
```

6 Conclusions and Future Work

In this deliverable we have described the use of a diagnostic visualiser for the debugging of interaction models and their executions. We have described the module that each kernel must possess if it wishes to display this information. We

Algorithm 3 EXTRACTVAR($trans$)

```
    target.var ← source.var
 2: for var ∈ trans ∧ isFreeInTrans(var, trans) do
        target.var ← target.var + addToLcc(var)
 4: end for
    for var ∈ trans ∧ isBoundInIllocution(var, trans) do
 6:     target.var ← target.var + checkBindingInLcc(var)
    end for
 8: return target.var
```

have described the process by which peers subscribe to receive process execution information from the controlling coordinator before receiving updates as and when the interaction proceeds.

We have also described early work on the creation of visual composition tools that allow the graphical creation of interaction models. In particular, we have described how the existing user interface components that are available for electronics institutions can be utilised for the creation of LCC protocols by a translation algorithm that can process the output of the Islander graphical schema.

We plan to provide a user interface module, such as Islander, for composing protocols as part of the standard distribution package. Ideally, this user interface module should be able to both allow creation of protocols as well as visualisation of the execution of the protocols that the diagnostic module is able to provide. That way, the user is provided with a consistent suite of tools that lead them from the creation process of an interaction, through the debugging process, to the actual execution of a protocol on the OpenKnowledge network.

References

[1] J. Ll. Arcos, M. Esteva, P. Noriega, J. A. Rodrìguez-Aguilar, and C. Sierra., *Engineering open environments with electronic institutions*, Engineering Applications of Artificial Intelligence, vol. 18.

[2] David Robertson, *Multi-agent coordination as distributed logic programming.*, ICLP, 2004, pp. 416–430.